Bienvenidos al Dojo

En este documento encontrareis un seguido katas (ejercicios). La primera kata que veremos a continuación está completamente guiada, solo debéis seguir los pasos que encontraréis.

Las siguientes debéis desarrollarlas vosotros mismos. En algunos de los ejercicios se sugieren tests a pasar pero no dudéis en añadir otros adicionales si lo considerais necesario para validar vuestro código. Intentad evitar redundancia y tests superfluos.

En caso de necesitar acceder a la documentación de Jest (el test runner) podéis encontrarla aquí.

Formato de entrega

A través del campus deberéis entregar un único fichero zip con el contenido de los archivos de desarrollo de las diferentes katas así como sus ficheros spec asociados. Similar a esta estructura pero para todas las katas. Si preferís hacer una carpeta por kata o alguna otro distribución que os resulte más cómoda adelante siempre que incluyáis la implementación y los tests de cada una de ellas.

Consultas

En caso de que tengáis que consultar algo mediante el foro o por correo electrónico copiad vuestro código a una de estas dos plataformas online y enviad el link del ejercicio para evitar problemas con el correo de la **UOC** que se carga los ficheros .js para evitar inyectar código malicioso y la tediosa mala indentación al copiar pegar el código en el correo electrónico. Usad cualquier de estos dos enlaces y la comunicación e interacción será mucho más sencilla:

- Codepen (para sencillos snippets de código)
- CodeSandBox (para ejercicios más complejos)

En caso de postear algún código en el foro que sean consultas genéricas y no directamente soluciones a los ejercicios. Hacer accesible soluciones de ejercicios a otros compañeros, aunque pueda no tener una intención directa, se considerará copia y se penalizará académicamente a nivel de asignatura.

Puntuación

El trabajar con TDD os permitirá a vosotros mismos tener una idea de vuestra propia nota antes de la entrega por lo tanto:

- **No se puntuará ejercicios que no incluyan tests**.
- Un ejercicio con tests que fallan tendrá una nota máxima de 5 y y será su optimización y corrección lo que determinará la puntuación entre 0 y 5.
- Un ejercicio que pase pasa los test indicados en el enunciado o, en caso de no estar indicados, los implementados por el alumno que incluyan todas las casuísticas necesarias para comprobar el correcto funcionamiento del código implementado tendrá como nota mínima un 5 y será su optimización y corrección en las formas lo que determinará la puntuación entre 5 y 10.
- Se valorará la legibilidad y sencillez del código sobre la sofisticación

Propiedad intelectual y plagio

La Normativa académica de la UOC dispone que el proceso de evaluación se cimenta en el trabajo personal del estudiante y presupone la autenticidad de la autoría y la originalidad de los ejercicios realizados.

La ausencia de originalidad en la autoría o el mal uso de las condiciones en las que se realiza la evaluación de la asignatura es una infracción que puede tener consecuencias académicas graves.

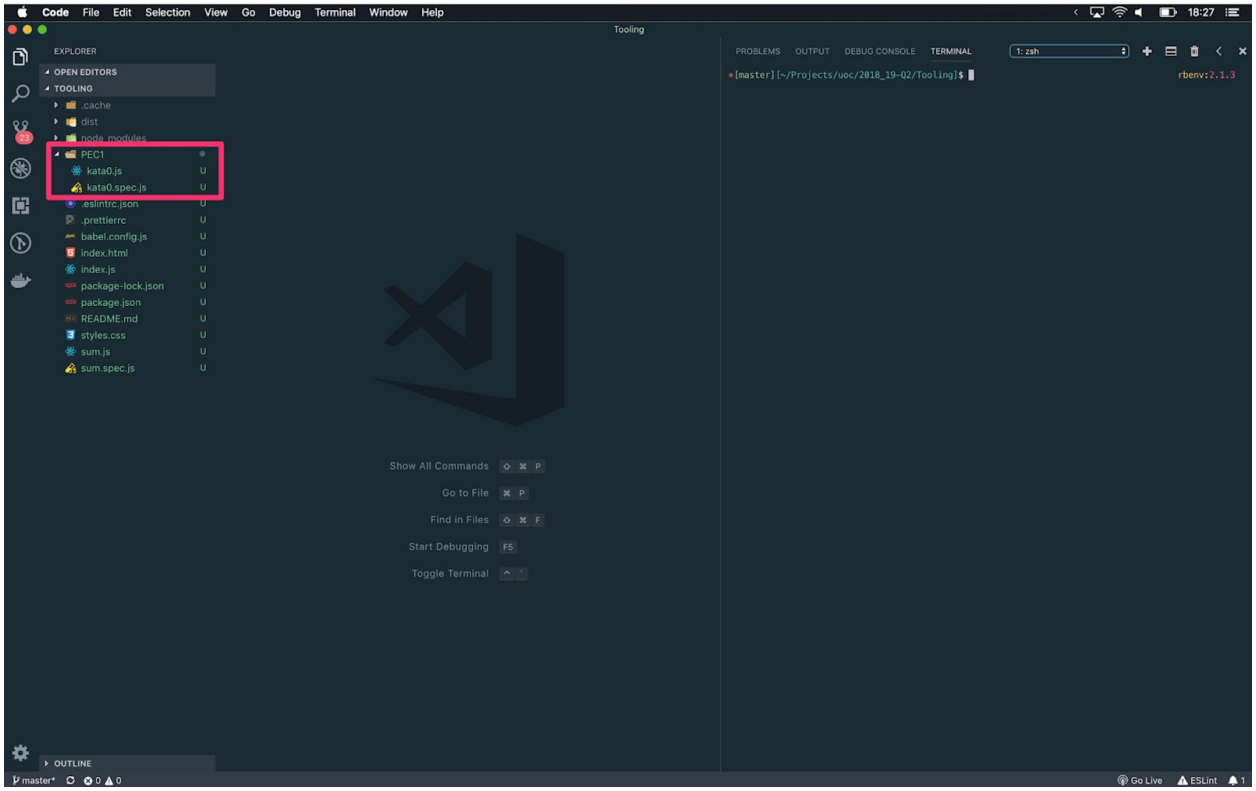
El estudiante será calificado con un suspenso (D/0) si se detecta falta de originalidad en la autoría de alguna prueba de evaluación contínua (PEC) o final (PEF), sea porque haya utilizado material o dispositivos no autorizados, sea porque ha copiado textualmente de internet, o ha copiado apuntes, de PEC, de materiales, manuales o artículos (sin la cita correspondiente) o de otro estudiante, o por cualquier otra conducta irregular.

Kata #0: sumatorio

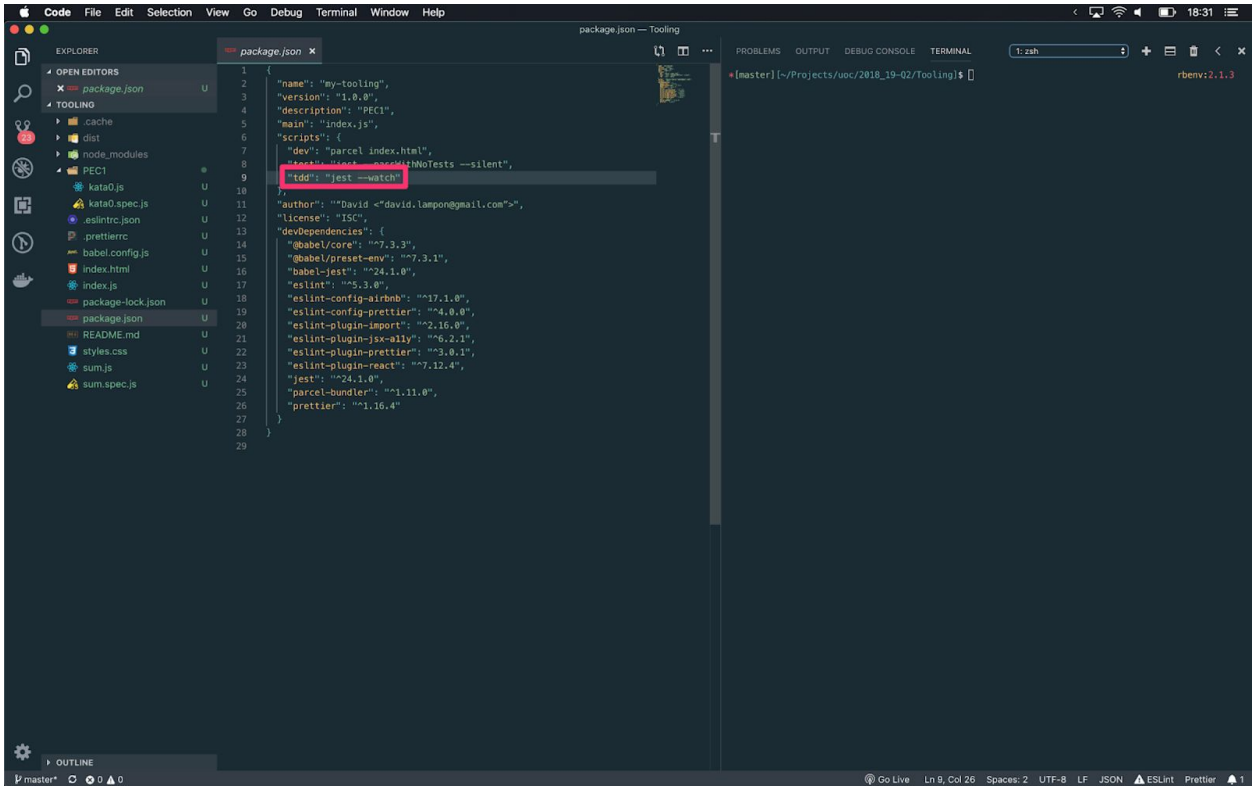
Haremos este primer ejemplo paso a paso para que os resulte más sencillo enfocar los que vienen a continuación:

Primero debemos entender lo que se nos pide: ¿qué es la operación **sumatorio**? Vamos a simplificar el sumatorio matemático general al caso puntual de sumar todos sus valores consecutivos hasta llegar a 0, siendo 1 el último valor a sumar. Por ejemplo: sumatorio de 3 equivale a 3 + 2 + 1 = 6.

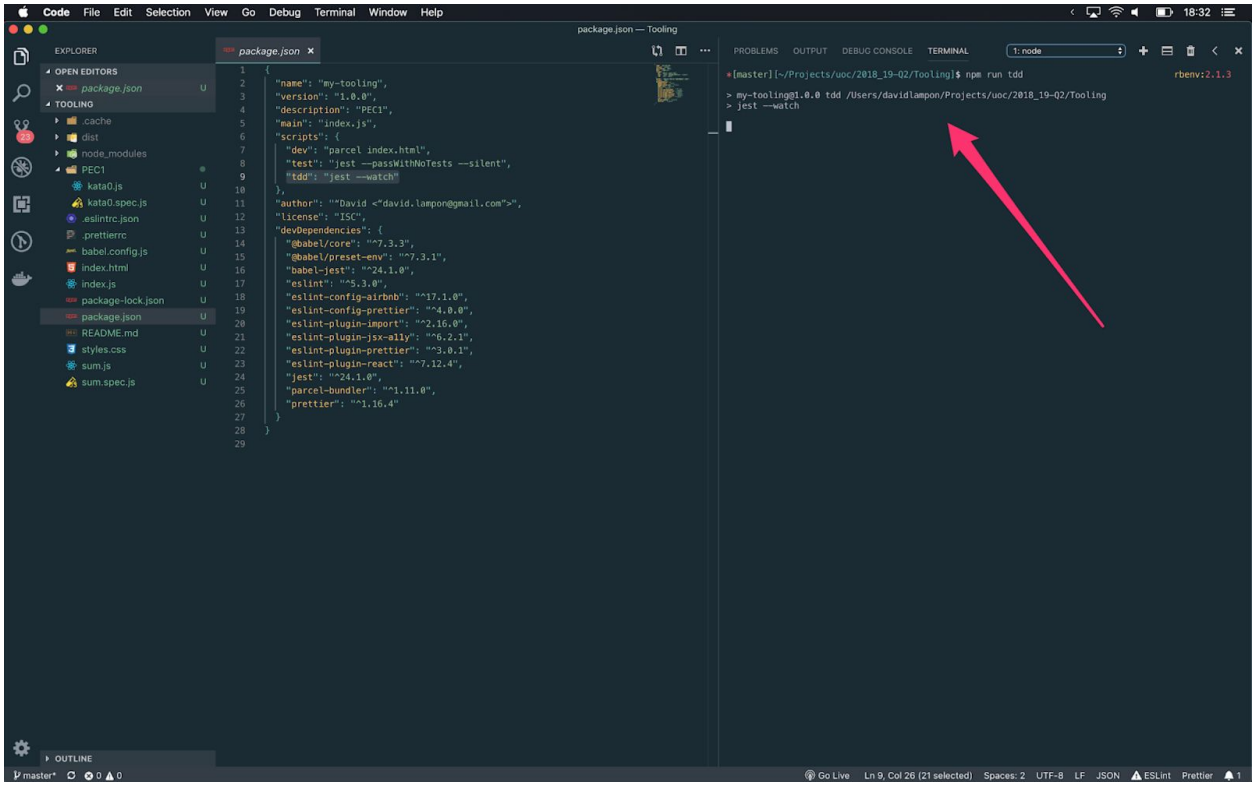
Abrimos nuestro proyecto de tooling y creamos una nueva carpeta para esta **PEC1** junto con dos archivos: **kata0.js** para la función sumatorio y **kata0.spec.js** para sus tests asociados.



Vamos a añadir un nuevo **script** para poder llevar a cabo tdd. Añadiremos estas lineas a nuestro fichero **package.json**: “*tdd": "jest --watchAll"*,



Este nuevo comando: *"tdd": "jest --watchAll"* nos permitirá arrancar **jest** en modo **watch** para que vaya pasando los tests cada vez que guardemos un fichero. Vamos a la consola y ejecutamos *npm run tdd*.



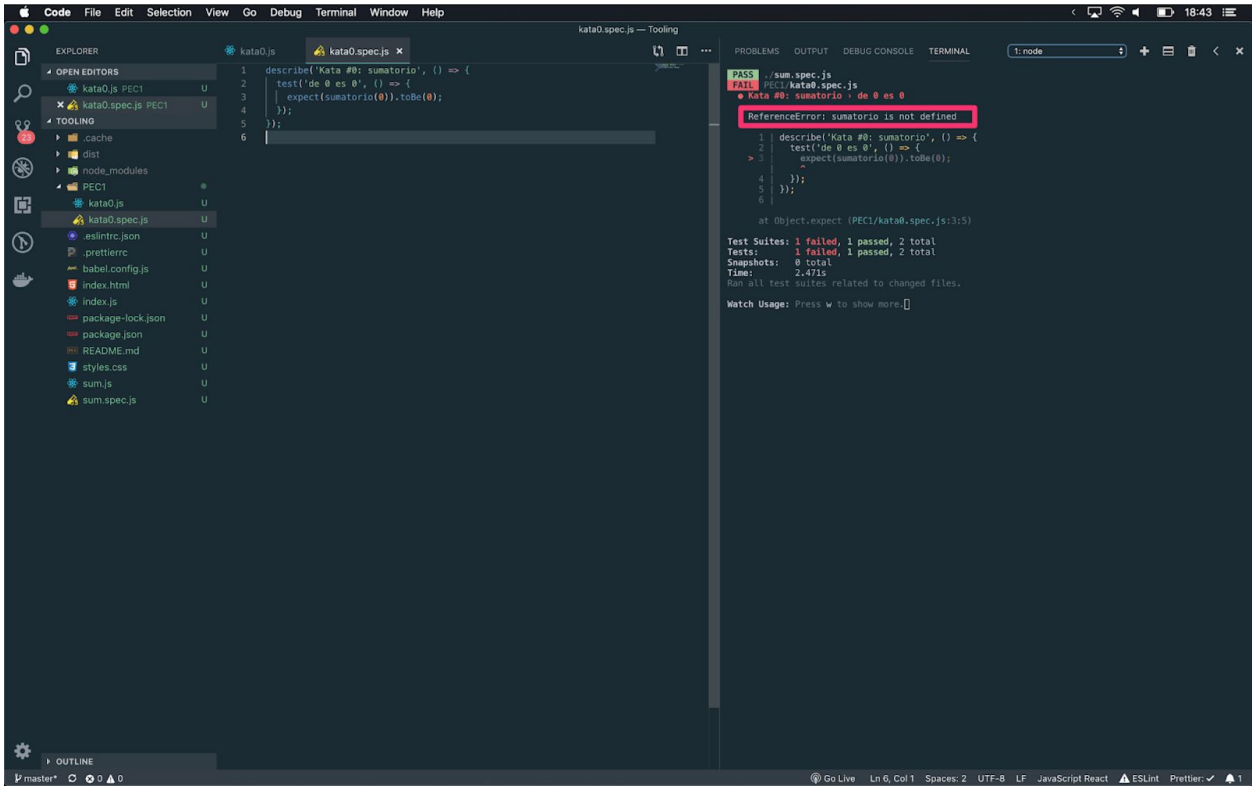
Jest se quedará en modo de espera a que empecemos a trabajar nuestros tests.

Ahora vamos a realizar un esfuerzo mental contra intuitivo. Vamos a implementar el primer test antes de siquiera tener la función. Como comentamos en la primera actividad TDD requiere que todos los tests escritos fallen antes de ser válidos en una siguiente iteración para verificar que cuando hay un error el test no pasa. En caso de no hacerlo así nunca sabríamos si los tests funcionan correctamente.

Este será nuestro primer test:

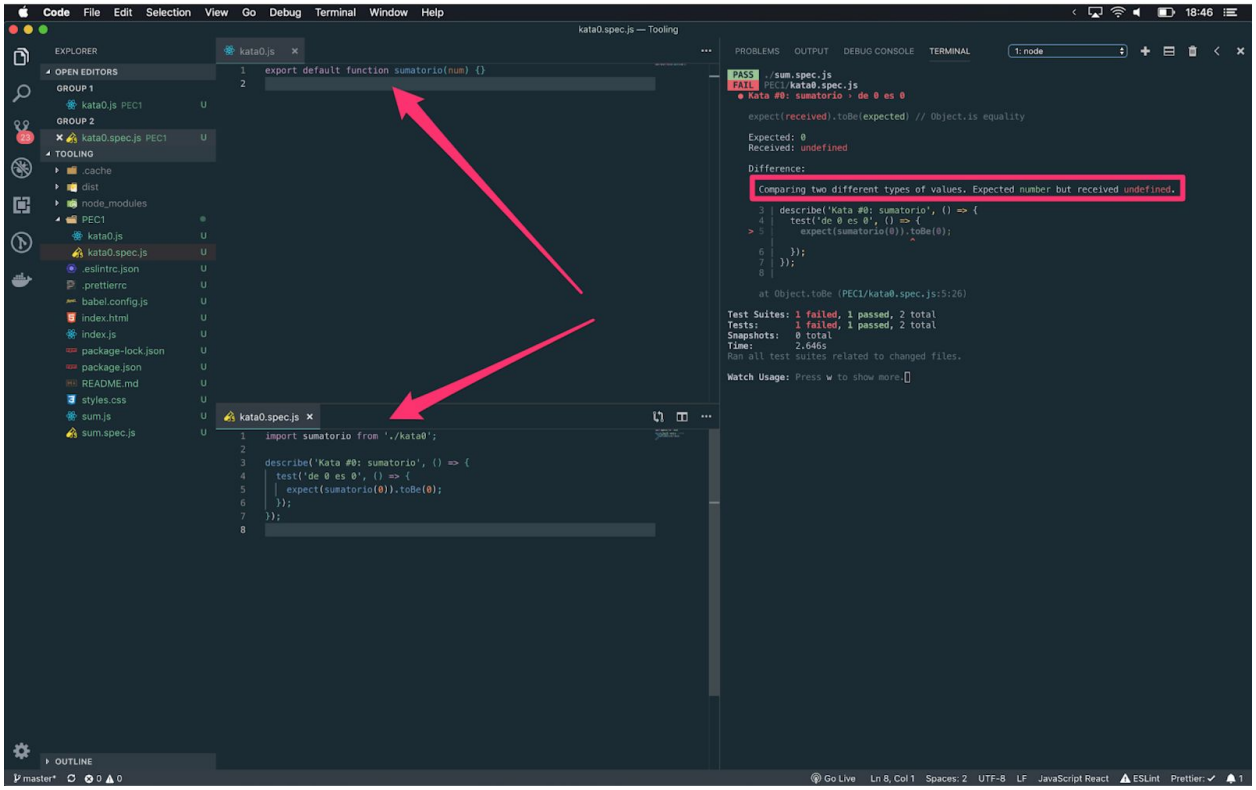
```
describe('Kata #0: sumatorio', () => {
    test('de 0 es 0', () => {
        expect(sumatorio(0)).toBe(0);
    });
});
```

Con el bloque describe creamos un bloque que agrupa varios tests relacionados, en nuestro caso todos los de la kata #0. Con el bloque *test* definiremos cada una de las condiciones que debe cumplir nuestra función, en primer lugar tomaremos el caso más básico que es que el sumatorio de 0 debe ser 0.



El mensaje que recibimos obviamente es que la función sumatorio no está definida. Lógicamente este caso iba a fallar pero está bien cometer este error para asegurarnos que a nivel estructural el código es correcto.

Ahora que tenemos un test que falla pasamos a corregirlo en la siguiente iteración:



Definimos la función sumatorio que toma un parámetro num y la exportamos como default del módulo. Se trata de la firma de la función, de momento sin ninguna implementación:

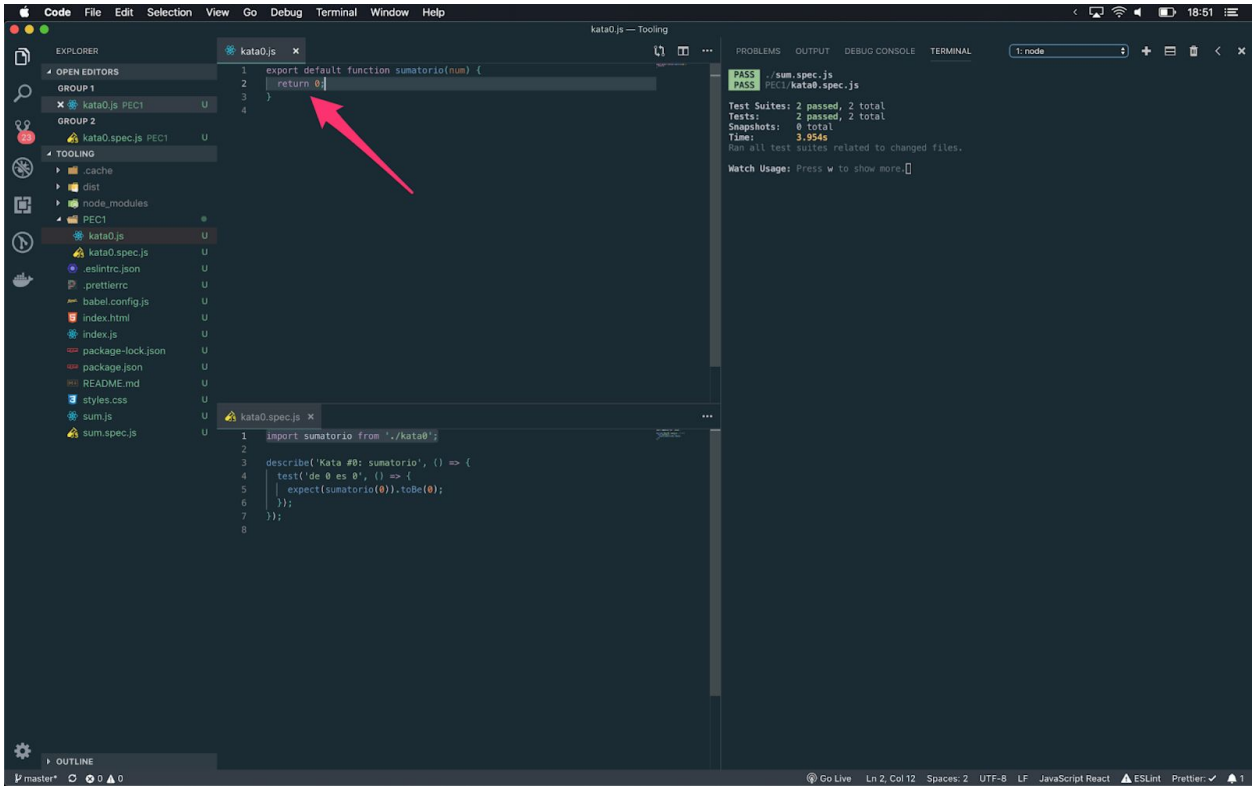
```
export default function sumatorio(num) {}.
```

En el fichero de test importamos la función que acabamos de definir:

```
import sumatorio from './kata0';
```

Ahora el error que recibimos es diferente, ya no nos habla del problema de una función indefinida sino que dicha función existe pero que no pasa el primer test.

La metodología TDD define que para desarrollar debemos implementar el código más sencillo que hace que el test se cumpla. En nuestro caso sería algo similar a esto:



Estoy seguro de que vais a pensar que es de perogrullo y que ese caso no cubre para nada lo que debe hacer la función sumatorio pero TDD implica este ciclo:

- escribir un test que falle
- escribir el mínimo código posible para que pase el test sin romper los anteriores
- iterar y refactorizar para adecuar el código

El siguiente test será:

```
test('de 1 es 1', () => {
    expect(sumatorio(1)).toBe(1);
  });
```

Obviamente este test no queda cubierto por nuestro código anterior así que añadimos el mínimo código para hacer que se cumpla:

```
export default function sumatorio(num) {
  return num;
}
```

En este caso sabemos que los resultados son casualmente correctos en base a los argumentos recibidos de la función. Estamos seguros de que no habrá más casos fantásticos como este (posiblemente) pero hemos seguido el protocolo de TDD una vez más. Los tests vuelven a pasar.

Añadimos un tercer test (el primero que anticipamos realmente interesante):

```
test('de 2 es 3', () => {
    expect(sumatorio(2)).toBe(3);
  });
```

Ahora tendremos que repensar un poco más la función porque el test no pasa (en caso de pasar sería redundante):

```
export default function sumatorio(num) {
  if (num === 2) {
    return 3;
  }

  return num;
}
```

Vale. Estamos jugando sucio y lo sabemos. Hemos conseguido que los tests pasen pero nuestro código es demasiado estático. A partir de ahora ya podemos intuir que cualquier caso adicional va a requerir de un condicional y retornar un valor concreto que obviamente es un proceso que no escala ya que queremos que sea automático en base al argumento. Vamos a intentar refactorizar el código para cubrir los tres casos actuales antes de pasar al siguiente:

```
export default function sumatorio(num) {
  let sumatorio = 0;
  let valorActual = num;

  while (valorActual !== 0) {
    sumatorio = sumatorio + valorActual;
    valorActual = valorActual - 1;
  }

  return sumatorio;
}
```

Esta es una solución muy verbosa y legible de una pieza de código que cubre los tres casos. Conociendo el procedimiento matemático del sumatorio que hemos propuesto somos conscientes de que cualquier nuevo caso iba a quedar cubierto. Añadamos un nuevo test:

```
test('de 3 es 6', () => {
    expect(sumatorio(3)).toBe(6);
  });
```

Si pasamos los tests de nuevo vemos que queda cubierto con nuestro código actual por lo que no tiene sentido añadir algo redundante así que lo borramos.

Otras implementaciones más sofisticadas podrían ser ésta que abusa del algoritmo matemático en lugar de un proceder de casuísticas como ha sido nuestro desarrollo:

```
export default function sumatorio(num) {
  return (num * (num + 1)) / 2;
}
```

O ésta que usa recursividad:

```
export default function sumatorio(num) {
  return num ? num + sumatorio(num - 1) : 0;
}
```

Todas ellas válidas y con un rendimiento aproximadamente igual. Sin embargo a mi humilde entender, la primera tiene una gran ventaja y es que no hace falta pensar mucho para entenderla lo cual es una gran ventaja cuando se trabaja en equipo y hay que mantener una base de código grande. Contra más sencillo y legible sea el código, mejor. Al menos, como reitero, desde mi muy personal punto de vista.

Este desarrollo nos da la seguridad que al trabajar con código modularizado esta parte siempre va a funcionar correctamente y si en algún momento debemos modificarla o ampliarla ya tenemos los tests que nos indicarán si los cambios introducidos rompen la plataforma.

Soy de la opinión de que muchos tests son innecesarios una vez tenemos práctica suficiente pero como modelo y mecanismo mental es muy interesante exponeros a este paso a paso en las siguientes katas. Llegó vuestro turno.

Kata #1: par o impar

Implementa la función **par_o_impar(Number)** que toma un entero como argumento y devuelve “Par” para números pares o “Impar” para números impares.

Debe cumplir estos tests:

- **par_o_impar(0)** devuelve el string “Par”
- **par_o_impar(1)** devuelve el string “Impar”
- **par_o_impar(2)** devuelve el string “Par”
- **par_o_impar(3)** devuelve el string “Impar”

Kata #2: suma de los elementos positivos de una matriz

Implementa la función **suma_de_elementos_positivos(Array de Numbers)** que toma una array de enteros como argumento y devuelve la suma total de los elementos positivos.

Debe cumplir estos tests:

- **suma_de_elementos_positivos([])** devuelve el número 0
- **suma_de_elementos_positivos([1,2,3,4,5])** devuelve el número 15
- **suma_de_elementos_positivos([1,-2,3,4,5])** devuelve el número 13
- **suma_de_elementos_positivos([-1,2,3,4,-5])** devuelve el número 9
- **suma_de_elementos_positivos([-1,-2,-3,-4,-5])** devuelve el número 0

Kata #3: repite un string

Implementa la función **repiteString(String, Number)** que toma un string y un número como argumentos y devuelve un string que contiene el string argumento repetido el número de veces designado por el argumento numérico.

Debe cumplir estos tests:

- **repiteString(‘JavaScript’, 0)** devuelve el string ‘’
- **repiteString(‘miau’, 1)** devuelve el string ‘miau’
- **repiteString(‘hola’, 3)** devuelve el string ‘holaholahola’
- **repiteString(‘?’, 10)** devuelve el string ‘??????????’

Kata #4: elimina el primer y último carácter

Implementa la función **elimina_primer_y_ultimo(String)** que toma un string como argumento y devuelve el string argumento pero sin su primer y último carácter, obviando los strings con 2 caracteres o menos.

Debe cumplir estos tests:

- **elimina_primer_y_ultimo('JavaScript')** devuelve el string 'avaScrip'
- **elimina_primer_y_ultimo('Alejandría')** devuelve el string 'lejandrí'
- **elimina_primer_y_ultimo('hidrógeno')** devuelve el string 'idrógen'
- **elimina_primer_y_ultimo('ok')** devuelve el string 'ok'

Kata #5: elimina los espacios

Implementa la función **elimina_los_espacios(String)** que toma un string como argumento y devuelve el string argumento sin espacios.

Debe cumplir estos tests:

- **elimina_los_espacios('buenos días')** devuelve el string ' buenosdías'
- **elimina_los_espacios(' pastel de zanahoria ')** devuelve el string 'pasteldezanahoria
- **elimina_los_espacios('dábale arroz a la zorra el abad')** devuelve el string 'dábalearrozalazorraelabad'

Kata #6: scope y closure

Nota: no son necesarios tests para esta kata

En este ejercicio no debéis implementar ningún código a no ser que queráis verificar vuestra respuesta. La solución a este ejercicio pasa por una explicación redactada por vuestra parte, con vuestras palabras sobre lo que sucede en este código y el resultado de cada una de las líneas ejecutadas.

No espero una explicación perfecta a nivel académico sino comprobar que vuestro entendimiento del código es correcto. Por supuesto, podéis implementar el código y ejecutarlo para corroborar vuestras respuestas.

```
1 function test() {
2   console.log(a);
```

```
3   console.log(foo());

4   var a = 1;
5   function foo() {
6      return 2;
   }
}

7 test();
```

Kata #7: this

Nota: no son necesarios tests para esta kata

```
var coche = {
  marca: "Ford",
  obtenerMarca: function(){
    console.log(this.marca);
  }
};

coche.obtenerMarca(); // Ford

var marcaDelCoche = coche.obtenerMarca;

marcaDelCoche(); // undefined
```

En este ejercicio estamos definiendo un objeto con un método que podemos invocar como **coche.obtenerMarca()** y obtener su valor.

Más tarde asignamos ese método a una variable. Qué hay que hacer para que la última línea devuelva correctamente el valor “Ford”? Razona tu respuesta.

Kata #8: objetos

Kata #8.1

Implementa el objeto **arbol** con las propiedades **especie** de valor **manzano** y **fruta** de valor **manzana**.

El objeto **arbol** debe cumplir estos tests:

- Debe tener una propiedad **especie** y una **fruta**
- El valor de la propiedad **especie** debe ser **manzano**
- El valor de la propiedad **fruta** debe ser **manzana**
- Debe existir un método **obtenerFruta**
- El resultado de invocar el método **obtenerFruta** debe ser **manzana**

Nota: me gustaría hacer hincapié en lo similar que es el enunciado a la definición de los tests. Es por esto que un buen conjunto de tests definen exactamente lo que hace y cómo funciona el módulo.

Kata #8.2

Implementa la función **obtenerFruta** que recibe un objeto y devuelve el valor de la propiedad **fruta**.

Debe cumplir estos tests:

- El elemento recibido tiene la propiedad **fruta**
- Si el elemento recibido tiene la propiedad **fruta** devuelve su valor (usad el objeto **arbol** para que devuelva **manzana**), en cualquier otro caso devuelve el string **“No fruta”**

Kata #9: factoría de objetos

Como hemos visto en el ejercicio anterior obtener la fruta parece una operación intrínseca al objeto **arbol**. Es cierto que podemos obtener directamente la propiedad del objeto pero si queremos implementar ofuscación y privacidad de datos debemos tener a mano métodos que accedan y manipulen los datos como nosotros (propietarios de los datos y los métodos) queremos que se reciban.

Kata #9.1

Implementa la función **plantarArbol** que se invoca pasándole dos argumentos (especie y fruta) y devuelve un objeto con esos valores para dichas propiedades. Esto es básicamente un constructor o una factoría.

Debe cumplir estos tests:

- En caso de no recibir dos parámetros **string** del tipo string devolverá **null** (esto son varias comprobaciones)
- Al invocar la función con los parámetros **peral** y **pera** devuelve el objeto:

```
{
    especie: 'peral',
    fruta: 'pera'
}
```

Pistas:

- https://jestjs.io/docs/en/expect#tobenull
- https://developer.mozilla.org/es/docs/Web/JavaScript/Referencia/Operadores/typeof

Kata #9.2

Al haber creado una función para crear y devolver objetos hemos construido un closure alrededor de nuestros datos por lo que podemos asignarlos al scope propio de la función.

En la implementación de la función en lugar de devolver directamente los parámetros recibidos vamos a implementar una nueva propiedad (método) en el objeto devuelto que sea la función **obtenerFruta** que lo que hace es, obviamente, devolver **fruta**.

Debe cumplir estos tests (es posible que tengáis que modificar alguno previo):

- El objeto devuelto incluye tres propiedades: **especie**, **fruta** y **obtenerFruta** (esto son tres comprobaciones independientes)
- Al invocar el método **obtenerFruta** nos devuelve el valor **pera**

Con esto comprobamos cómo gracias a un método podemos enmascarar el valor de la propiedad al obtenerla mediante un método. Esto nos permite tener privacidad de datos ya que desactivamos el acceso directo a los datos. Estos métodos que nos permiten acceder a propiedades se conocen como getters.

Kata #9.3

Implementa por tu cuenta el getter para **especie**: **obtenerEspecie.** Elimina del objeto devuelto las propiedades **fruta** y **especie** y cambialos por los getters **obtenerFruta** y **obtenerEspecie.**

Debe cumplir estos tests:

- El objeto devuelto incluye dos propiedades: **obtenerEspecie** y **obtenerFruta**
- Al invocar el método **obtenerEspecie** nos devuelve el valor **peral**
- Al invocar el método **obtenerFruta** nos devuelve el valor **pera**

Kata #9.4

Es posible que los getters no den una percepción total de la potencia de haber implementado privacidad de datos. Por ello vamos a verlo implementado los setters. Vamos a implementar y devolver en el objeto dos métodos más: **definirEspecie** y **definirFruta.** Es obvio su propósito y su implementación pero su potencia radica en que al no acceder directamente a los parámetros podemos hacer un control de errores.

Debe cumplir estos tests:

- El objeto devuelto incluye cuatro propiedades: **obtenerEspecie, definirEspecie** y **obtenerFruta** y **definirFruta**
- Al crear un arbol e invocar posteriormente **definirFruta** con el valor **12** el objeto **arbol** mantiene su valor previo (*volveremos a este punto cuando estemos viendo gestión de errores*)
- Al crear un arbol e invocar posteriormente **definirFruta** con el valor **'manzana'** el objeto **arbol** tiene **manzana** como valor la propiedad **fruta**.
- Haced el equivalente del superior para **definirEspecie**

Pista: es posible que necesiteis definir variables internas **var** o **let** (reasignables) para poder hacer la asignación inicial y la posterior reasignación de valores.

Kata #10: Prototipo

La potencia del prototipo en JavaScript es enorme. Nosotros vamos a hacer un caso sencillo y bastante insustancial pero que nos servirá para entender cómo podemos extender la funcionalidad.

Podríamos hacer una extensión de nuestro propio módulo pero resulta poco concluyente cuando tenemos acceso directo a su implementación. Sin embargo imaginad que queremos añadir una funcionalidad muy concreta sobre cada instancia del tipo **String** en nuestra aplicación. Vamos a añadir un método para que dado un string cualquiera podamos invocar el método **presentarArbol** que lo único que hará es un console.log del string sobre el que se ejecuta más el texto

```
"Este árbol es un ${nuestro_arbol} y da ${nuestra_fruta}".
```

Debéis modificar el prototipo de String dentro o fuera de la función pero dentro del módulo y luego pasar los siguientes tests:

- **arbol** tiene el método **obtenerEspecie** que devuelve un string que tiene el método **presentarArbol**

- Al ejecutar **presentarArbol** se presentar por consola “"Este árbol es un ${nuestro_arbol}”, siendo **nuestro_arbol** el string sobre el que se ejecuta el método **presentarArbol**
- Al ejecutar **presentarArbol** con un parámetro de tipo **string** se presentar por consola “"Este árbol es un ${nuestro_arbol} y da ${nuestra_fruta}”, siendo **nuestra_fruta** el string que hemos pasado al método

Pista:
https://developer.mozilla.org/es/docs/Web/JavaScript/Herencia_y_la_cadena_de_protipos

Kata #11: Gestión de errores

A partir del caso anterior vamos a usar la manera habitual para lanzar un error JavaScript. Añadid estas líneas en la comprobación de tipo en el setter **definirFruta**.

```
throw new Error();
```

Debe cumplir estos tests:

- Al crear un arbol e invocar posteriormente **definirFruta** con el valor **12** el método lanza un error y mantiene su valor previo

Pista: https://jestjs.io/docs/en/expect#tothrowerror

Kata #12: Expresiones regulares

El uso de expresiones regulares es habitual en JavaScript para detectar patrones en cadenas de texto. Para nuestros ejercicios vamos a tomarnos ciertas licencias para generalizar el nombre de la especie y la fruta. Antes de validar un cambio mediante setters vamos a validar la fruta con respecto a la especie. Como ejemplo:

- manzano - manzana OK
- peral - pera OK
- naranjo - naranja OK
- nogal - nueces KO

De los tres primeros casos vemos que se repiten las 4 primeras letras de la especie también en la fruta. Ésta será la comprobación que haremos en nuestro setter.

Debe cumplir estos tests:

- Al crear un **arbol** con los valores **manzano** y **manzana** e invocar posteriormente **definirFruta** con el valor **pera** el método lanza un error y mantiene su valor previo

- Al crear un **arbol** con los valores **peral** y **manzana** e invocar posteriormente **definirFruta** con el valor **pera** el método modifica el valor de la propiedad **fruta**

Pistas:

- Creación de RegEx (usad el método que prefirais): https://developer.mozilla.org/es/docs/Web/JavaScript/Guide/Regular_Expressions#Creaci%C3%B3n_de_una_expresi%C3%B3n_regular
- Probador de regex: https://regexr.com, https://regex101.com/

Kata #13: Clases

Implementa todo lo anterior pero con el formato de clases. Todos los tests deben pasar. El nuevo código es en realidad solo una restructuración de todo lo hecho hasta ahora, con el formato de definición de las clases de ES6 y el constructor como método de llamada inicial al instanciar.

Pista: https://developer.mozilla.org/es/docs/Web/JavaScript/Referencia/Classes